Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 52 (2004) 341–370

Science of Computer Programming

www.elsevier.com/locate/scico

A tour of Tempo: a program specializer for the C language

Charles Consel[a,*], Julia L. Lawall[b], Anne-Françoise Le Meur[a,b]

[a] *Compose group, INRIA/LaBRI, ENSEIRB, 1 avenue du docteur Albert Schweitzer, 33402 Talence Cedex, France*

[b] *Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark*

Received 16 April 2003; received in revised form 7 November 2003; accepted 22 January 2004

Available online 4 June 2004

Abstract

Specialization is an automatic approach to customizing a program with respect to configuration values. In this paper, we present a survey of Tempo, a specializer for the C language. Tempo offers specialization at both compile time and run time, and both program and data specialization. To control the specialization process, Tempo provides the program developer with a declarative language to describe specialization opportunities for a given program.

The functionalities and features of Tempo have been driven by the needs of practical applications. Tempo has been successfully applied to a variety of realistic programs in areas such as operating systems and networking. We give an overview of the design of Tempo and of its use in specializing realistic applications.

Keywords: Program specialization; Data specialization; Program analysis; Imperative language; Applications

1. Introduction

Customizing programs for a specific usage context is a common optimization technique for computation-intensive applications. Customization is typically performed by the programmer and aims to eliminate genericity. The values of configuration parameters

* Corresponding author.

E-mail addresses: consel@labri.fr (C. Consel), julia@diku.dk (J.L. Lawall), lemeur@labri.fr, lemeur@diku.dk (A.-F. Le Meur).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.03.011

are propagated throughout the program and program fragments that manipulate these values are simplified. The transformation eliminates unneeded functionalities and performs configuration-time calculations. If successful, customization produces a program that is smaller or executes faster than the original one. Successful customizations have been reported in a number of areas: in computer graphics, Locanthi customizes a bit block transfer function (bitblt) [44]; in operating systems, Massalin and Pu systematically use customization to optimize systems components [51]; in scientific computing, it is a common practice to develop customized versions of generic libraries. Yet, because these approaches rely on manual transformation, they trade efficiency for safety and maintainability.

Program specialization is an approach to automating the customization process. To make customization automatic, program specialization provides a fixed and well-understood set of transformations commonly used in manual customization, including loop unrolling, aggressive constant propagation, and simplification of conditionals. Extensive research and experimentation have demonstrated the benefits of program specialization in a variety of areas such as operating systems, networking, graphics, scientific computing, and compiler generation [11,29,53]. Historically, program specialization has been carried out at compile time and has performed a source-to-source transformation [30].

The first practical work on program specialization was that of Jones et al. in the early 1980's [30]. In the years that followed, researchers actively explored this technique in a number of directions without challenging its compile-time and source-to-source nature. Nevertheless, in some application areas the values of configuration parameters only become available at run time. More than 10 years later, various approaches to specializing programs at *run time* were proposed [14,38,42], opening a number of new opportunities for specialization. Another extension to program specialization is to specialize a program in *multiple stages*, i.e., incrementally [20,50]. It was shown that, by factorizing the transformation process, specializing a program at each stage costs considerably less than specializing it only when all of the configuration values are available. Multi-stage specialization can be performed both at compile time and at run time.

The dual notion to specializing programs is specializing data; *data specialization* encodes the results of early computations in data structures [2,47], instead of encoding them in a residual program. Program and data specialization are complementary: program specialization can be used if the number of results to encode in the specialized program does not cause code blow-up; otherwise, data specialization can compactly represent results in data structures. Chirokoff et al. investigate the benefits and limitations of integrating both strategies into a single specialization process [7].

Initially, the main target application for program specialization was compiler generation from executable language specifications written in a denotational style [30]. As a result, early research efforts primarily focused on functional languages. The first program specializer, called Mix, processed untyped, first-order, side-effect free functional programs [30]. Many program specializers succeeded Mix and proposed new analyses and transformations to cope with higher-order functions and data structures [5,9,15,48]. A variety of other languages were explored ranging from logic to imperative languages [6,11,18,23,29,32,49,54]. As program specialization became more mature, research targeted widely used languages such as Fortran [33], C [13,29] and Java [69].

This paper

This paper gives a tour of a specializer for the C language, named Tempo. A key feature of Tempo is that it offers all of the main forms of specialization: it specializes programs both at compile time and run time; run-time specialization can be performed incrementally; Tempo offers both program and data specialization; finally, it has been used as a back-end to specialize programs written in other languages, namely, Java and C++. Interestingly, all of these forms of specialization share the same analysis phase.

Most previous program specializers have been targeted towards compiler generation. In contrast, the design of Tempo was targeted towards the needs of applications in systems and networking [12,53,58]. To cope with such applications, its design and implementation were iterated until it successfully optimized a set of representative programs. This process motivated the introduction of new analysis features, such as *return sensitivity* [24–26]. The resulting program specializer significantly improved the performance of industrial-strength code such as the remote procedure call developed by Sun [55].

As for any automatic program transformation tool, it is essential to make the use of a specializer convenient for programmers. Usability concerns include the ability to specify and to obtain the desired degree of specialization. To address these issues, we have extended Tempo with a declarative language that allows the programmer to specify specialization scenarios for a given program [41]. These scenarios are used to configure the specialization process and are checked during the analysis phase of Tempo to ensure that the result of specialization will match the programmer's expectations.

Working example

We illustrate our tour of Tempo using the remote procedure call (RPC) mechanism [3, 72]. The RPC is used to support the implementation of distributed services between heterogeneous machines. There have been many efforts to manually optimize critical parts of the RPC, to reduce, or even to eliminate, the overhead of this mechanism whenever possible [27,66]. In this paper, we focus on the marshaling of values in the RPC, which is carried out by the XDR library [71]. This library converts values into a machine independent format. Arguments of a remote procedure are marshaled by a client before being sent, and unmarshaled by the server when received. The procedure call result is similarly (un)marshaled.

The XDR implementation is well suited to illustrate program specialization because it consists of a set of highly generic micro-layers. Each layer is devoted to a small task, for example, marshaling some parameter or writing the result in memory. Roughly, a generic function implements each layer; each function interprets its arguments to determine the function to invoke in the layer below.

Let us illustrate the layered architecture of the XDR library by considering the marshaling of an integer value as an argument to a remote procedure. The associated call chain is summarized in Fig. 1, where the remote procedure call argument is stored in the variable `arg`. We assume that the configuration parameters of the XDR process have been stored in an *XDR state*, `xdrs`, higher in the call chain. In particular, the XDR state indicates that marshaling rather than unmarshaling is required.

```
xdr_int(xdrs, &arg)                  // Machine dependent switch on integer size
  xdr_long(...)                      // Generic marshaling or unmarshaling
    XDR_PUTLONG(...)                 // Generic marshaling to memory, stream, etc.
     xdrmem_putlong(...)             // Check for overflow and write to output buffer
       htonl(...)                    // Choice between big and little endian
```

Fig. 1. The abstract trace of the marshaling of an integer value.

```
bool_t xdrmem_putlong(XDR * xdrs, long * lp) {     // xdrs points to the XDR state
                                                   // and lp points to data
   if ((xdrs->x_handy -= sizeof(long)) < 0)        // Decrement space left in buffer
         return (FALSE);                           //     Return failure on overflow
   *(long *)xdrs->x_private = htonl(*lp);          // Copy to buffer
   xdrs->x_private += sizeof(long);                // Point to next copy location
   return (TRUE);                                  // Return success
}
```

Fig. 2. The source code of the `xdrmem_putlong` function.

Initially, the function `xdr_int` is invoked with the XDR state and the address of the integer value to be marshaled. Because integers are represented as long integers on the host machine, this function invokes `xdr_long` to further process the value. The function `xdr_long` examines the XDR state to determine the coding direction (i.e. marshaling or unmarshaling) and calls the function `XDR_PUTLONG` in this case to perform the marshaling. The XDR state is further examined to determine whether the integer value should be stored in memory or in a stream. The former option is taken and so the function `xdrmem_putlong` is invoked. This function uses `htonl` to change the order of the bytes, if needed, and then writes the marshaled value into a buffer.

To further explore the marshaling process, let us examine the definition of `xdrmem_putlong`, displayed in Fig. 2. This function takes a state of type `XDR` and a pointer to the data to be marshaled. The XDR state is used to determine the amount of space left in the output buffer (the field `x_handy`) and to obtain a pointer to the current position in this buffer (the field `x_private`). If the value of the `x_handy` field indicates that there is enough space left, the function `htonl` is invoked to produce an integer value with an appropriate byte ordering.

We can specialize `xdr_int` with respect to the coding direction and the current position in the output buffer. Specialization unrolls the function calls down to the function `xdrmem_putlong`. At this point, specialization removes the overflow check and the pointer increment performed by `xdrmem_putlong`, leaving only the buffer copy, which depends on the value to marshal, and the return value. The result of this specialization is shown in Fig. 3.

Overview

The rest of this paper is organized as follows. Section 2 describes the anatomy of Tempo, examining the analyses that are performed and the various forms of specialization that build on the results of these analyses. Section 3 lists some applications successfully customized

```
bool_t xdr_int_spe (XDR *xdrs, int *ip) {
  (long *)xdrs->x_private = htonl(*ip);
  return TRUE;
}
```

Fig. 3. A specialized instance of `xdr_int`.

by Tempo and presents some performance measurements. Section 4 presents some initial work in improving the usability of Tempo, including specialization declarations, verification of these declarations, and tools for visualizing analysis results. Section 5 describes some related work. Finally, Section 6 concludes and proposes future directions.

2. A tour of the Tempo specialization engine

Program specialization is typically performed in two phases: *preprocessing* and *processing*, as illustrated in Fig. 4. The preprocessing phase first identifies the computations that can be performed during specialization, based on the program and a description of the configuration parameters (parameters whose values are available at specialization time). From this information, the preprocessing phase then generates a dedicated specializer. The processing phase uses this dedicated specializer to customize the program with respect to the actual configuration values, reducing the computations identified as depending only on the configuration values and reconstructing (i.e., *residualizing*) the rest. Preprocessing is to be performed by the *program developer*, who is aware of the program structure and the specialization opportunities that the program provides. Processing is to be performed by any number of *users*, who apply the dedicated specializer to the specific configuration values that are appropriate to their needs.

The design of Tempo has been guided by the desire for a uniform approach to the different forms of specialization: compile-time and run-time program specialization, and data specialization. Thus, all three forms of specialization use the same analysis engine. We first present the analyses and then give an overview of the different forms of specialization.[1]

2.1. Analysis phase

Fig. 5 shows the sequence of analyses and transformations that are performed by Tempo. We focus on the last three analyses, *binding-time analysis*, *evaluation-time analysis*, and *action analysis*. Given the program and the description of available configuration parameters, these three analyses amount to a form of dependency analysis that classifies the computations that solely depend on the configuration parameters as *static* and the remaining computations as *dynamic*. The result of these analyses is an annotated program.

2.1.1. Binding-time analysis

Binding-time analysis identifies expressions that depend only on the values of configuration parameters and on other static information (e.g., constants) available in

[1] Tempo uses a subset of C as an internal representation. We have translated all of the figures in this section from this internal representation to the form of the original source program, for readability.

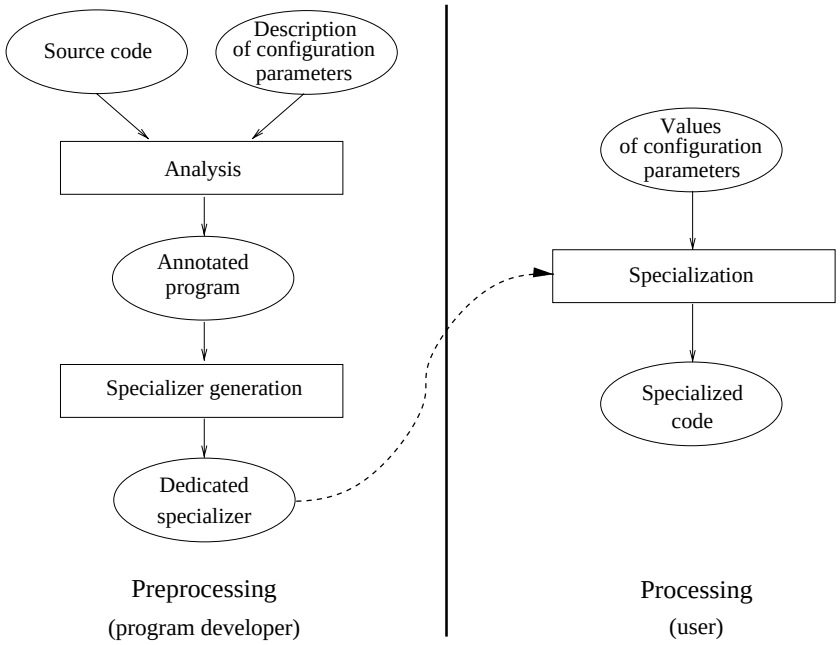

Fig. 4. Decomposition of the specialization process.

the program. We give only an overview of the analysis. Hornof and Noyé present the binding-time analysis of Tempo in more detail [25].

Tempo uses the binding times *static* and *dynamic*. The analysis is constructed such that any static expression can safely be reannotated as dynamic in the course of the analysis (technically, static $\sqsubseteq$ dynamic). An expression is dynamic if it is a variable that has been determined to be dynamic or if it has a dynamic subexpression. For example, if `s` is static and `d` is dynamic, then `d + (2 * s)` is dynamic. An expression is static otherwise. In the previous example, the subexpression `2 * s` is static.

A key point in the binding-time analysis of an imperative language is the flow of binding-time properties across assignments. Some representative cases are shown in Fig. 6. When the left-hand side of an assignment is a simple variable, the binding time of the variable becomes the binding time of the right-hand side expression. For example, in the code following the assignment `x = 3` in line (1), the variable `x` is static. When the left-hand side is a dereference expression, the effect of the assignment depends on the alias information. At the point of the assignment `*y = 4` on line (3), the alias analysis of Tempo determines that the only possible alias of `*y` is `x`. In this case, the alias is given the binding time of the right-hand side expression. In the assignment `*y = 7` on line (5), however, the alias analysis determines that `*y` is either an alias of `x` or of `b`. Nevertheless, only one of `x` or `b` is actually affected by the assignment at specialization time depending on the value of `a`. In this case, the binding time of each alias is set to the least upper bound of its current binding time, reflecting the possibility that the variable is not affected by the assignment,

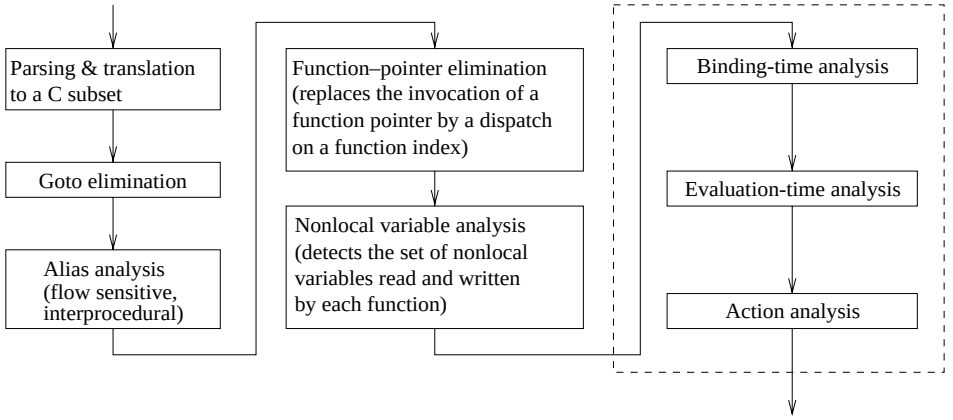

Fig. 5. Components of the analysis phase.

```
//  a is initially static,  b is initially dynamic
x = 3;                                          (1)
y = &x;                                         (2)
*y = 4;                                         (3)
if (a > 0) y = &b;                              (4)
*y = 7;                                         (5)
```

Fig. 6. An example to illustrate the binding-time analysis of assignments. The assignments in lines (1), (2), and (4) affect only the binding times of the left-hand side variables. The effect of the assignments in line (3) and (5) depends on the aliases inferred for `y`.

and the binding time of the right-hand side expression, reflecting the possibility that it is. Thus, in the example, the new binding time of `b` is the least upper bound of its original binding time (dynamic) and the binding time of `7` (static), which implies that `b` continues to be considered dynamic. Both possible binding times of `x` are static, so `x` remains static.

Binding times are affected both by data-flow dependencies, as in the examples above, and by control-flow dependencies, in the treatment of conditionals and loops (all gotos are encoded as conditionals and loops in the goto elimination phase shown in Fig. 5, following the algorithm of Erosa and Hendren [16]). When a conditional has a static test, only one of the branches is specialized and the values of any static variables assigned in the chosen branch are available to specialization of the rest of the code. Nevertheless, because the binding-time analysis cannot determine which branch is chosen during specialization, the binding time of each variable assigned within either branch (as determined by the alias analysis and the non-local variable analysis shown in Fig. 5) is set to the least upper bound of its binding times at the end of the two branches. For example, in Fig. 7(a), `x` is assigned a static value in both branches and is thus considered static after the conditional, while in Fig. 7(b), `x` is assigned a static value in one branch and a dynamic value in the other and thus is considered dynamic after the conditional. When a conditional has a dynamic test, both branches are specialized. In this case, a static variable modified in at least one of the branches potentially has two specialization-time values at the end of specialization of

```
// test is static
if (test) x = 3;
else x = 4;
... x ...      // x is static
```

(a)

```
// test is static, d is dynamic
if (test) x = 3;
else x = d;
... x ...      // x is dynamic
```

(b)

Fig. 7. An example to illustrate the binding-time analysis of conditionals. The effect of assignments in the branches of a conditional depends on the binding time of the test expression.

```
// x is initially dynamic
int x;

int f(int y) {
  return x + y;
}

...
f(4);                    // Variant of f with x dynamic, parameter static
f(x);                    // Variant of f with x dynamic, parameter dynamic
x = 3;
f(5);                    // Variant of f with x static, parameter static
```

Fig. 8. An example to illustrate the context-sensitive binding-time analysis of function calls. The analysis of the function `f` depends on the binding times of its arguments.

the conditional. Thus, all variables assigned in either branch are subsequently considered dynamic. For example, if the test expressions in Fig. 7 are dynamic, then the final reference to `x` is dynamic in both examples. Loops are treated by standard fix-point techniques [29].

To provide the degree of precision needed for realistic applications, the binding-time analysis of Tempo adopts several standard data-flow analysis strategies: *flow sensitivity*, *context sensitivity*, and *polyvariance* [25]. Flow sensitivity affects the treatment of assignments. Rather than maintaining a single binding time for all uses of a given variable, the binding time of a variable in Tempo depends on the most recent assignment of the variable or other binding-time effect (i.e., control dependence). For example, in Fig. 7(b), `x` is considered static in the "then" branch of the conditional, even though it is considered dynamic elsewhere in the program. Context sensitivity affects the treatment of function calls. Tempo creates a variant of a function for each binding-time configuration that occurs at any of the possible call sites, taking into account both the binding times of the function's parameters and the binding times of any variables referenced by the function, as determined by the non-local variable analysis phase (Fig. 5). Fig. 8 illustrates several calls to a function `f`, which references a global variable `x`. Suppose that `x` is initially dynamic. Each call to `f` involves a different set of binding times for the parameter and the variable `x`, and thus Tempo creates three variants of `f`. Finally, polyvariance of data structures, which is optional in Tempo, affects the treatment of structure fields. When polyvariance is used, each declared data structure is associated with a separate binding-time description. For example, in Fig. 9, the reference to `s1.a` in line (3) is considered dynamic, and the reference to `s2.a` in the same line is considered static. When polyvariance is not used, all instances of each structure type share the same description. Thus, in the same example, the initial assignment of the field of an `s`-typed structure to the dynamic value `d` (line (1))

```
// d is initially dynamic
struct s { int a; };
struct s s1;
struct s s1;

...
s1.a = d;                                                    (1)
s2.a = 3;                                                    (2)
return s1.a + s2.a;                                          (3)
```

Fig. 9. An example to illustrate the binding-time analysis of structures. With polyvariance, `s1` and `s2` are given separate binding times. Without polyvariance, all instances of structure type `s` are given the same binding time.

implies that all subsequent references to either `s1.a` or `s2.a` are considered dynamic. Polyvariant analysis is more precise, but is also more expensive and gives no extra benefit for some programs. Whether or not polyvariance is used, the binding-time description associated with a structure contains a separate binding time for each structure field.

Including flow-sensitive, context-sensitive, and polyvariant analysis features in the binding-time analysis of Tempo increases the availability of static information within a program at specialization time. Nevertheless, we found that these features were not sufficient to achieve the desired degree of specialization for many systems programs, such as the XDR library [58]. In systems code, it is common to return a status flag to indicate the success or failure of a computation, and the value of this flag often depends on the values of only a subset of the parameters. To address this issue, we developed the analysis feature *return sensitivity* for Tempo. When a function has a static return value, but contains dynamic computations, the return value is considered static at the call site, even though the function call must be residualized. An example is shown in Fig. 10. In the function `f`, the choice of return value only depends on the value of `s`, which is static based on the call to `f` in line (2). Because all of the possible return values are also static, the return value can be used in specializing the call site. Nevertheless, the call to `f` must still appear in the specialized program. In particular, its residual definition contains a specialized instance of the dynamic assignment on line (1).

In a realistic application, only a portion of the program may present interesting specialization opportunities. When specialization is applied to a program fragment, this fragment may call external functions that affect global variables that it subsequently references. For example, in Fig. 11, the function `f` calls the function `g`, which modifies the global variable `x`; this variable is subsequently referenced by `f`. If specialization is only applied to the fragment containing the declaration of `x` and the function `f`, then the specializer must be informed of the side-effect to `x` in the function `g`. To describe such side-effects, Tempo allows the program developer to create an *analysis context file* that describes the side-effects performed by external functions. Currently, the program developer provides C definitions of these functions that may abstract the original definition to the point that they only achieve the relevant binding-time effect, e.g. assigning dummy static or dynamic values to the affected variables rather than using the original right-hand side expressions. We are working on a more convenient notation for this information.

Finally, we consider binding-time analysis of the `xdrmem_putlong` function shown in Fig. 2, where the parameter `xdrs` is static, the parameter `lp` is dynamic, and the

```
// d is initially dynamic
int d;

int f(int s) {
  if (s == 0) return 0;
  d = d / s;                                                    (1)
  return 1;
}

...
if (!f(10)) {                                                   (2)
  ...  // Treat error case
}
...
```

Fig. 10. An example to illustrate the return-sensitive binding-time analysis of function calls. The static return value of `f` is available to subsequent specialization, even though parts of the body of `f` are dynamic.

```
// Code to be specialized

int x;

int f(int y) {
  g();
  return x + y;
}
```

```
// Code excluded from the
// specialization process

int g() {
  x = 12;
}
```

Fig. 11. An example to illustrate specialization of a program fragment. The function `f` to be specialized calls the function `g`, which is external to the specialized fragment.

```
bool_t xdrmem_putlong(XDR * xdrs, long * lp) {                  (1)
   if ((xdrs->x_handy -= sizeof(long)) < 0)                     (2)
         return (FALSE);                                        (3)
   *(long *)xdrs->x_private = (long)htonl((u_long)(*lp));       (4)
   xdrs->x_private += sizeof(long);                             (5)
   return (TRUE);                                               (6)
}                                                               (7)
```

Fig. 12. The result of binding-time analysis of `xdrmem_putlong` (dynamic constructs are underlined).

fields `x_handy` and `x_private` in the structure referenced by `xdrs` are static. The result is shown in Fig. 12. Dynamic constructs are underlined. Most of the computations in `xdrmem_putlong` are considered static. Only the reference to `lp` and the result of passing this value to `htonl` in the assignment of line (4) are considered dynamic. Even the left-hand side of this assignment, `*(long *)xdrs->x_private`, is considered static, because the address is known at specialization time. Furthermore, the subsequent reference to `xdrs->x_private` is static (line (5)) even though it now points to a dynamic value. Indeed, the precision of the binding-time analysis of Tempo is sufficient to support a static pointer to a dynamic value, which allows specialization-time computations to be performed on this pointer value, as shown here. Both values returned by this function are static (lines (3) and (6)). Return sensitivity allows specialization of the call site to use whichever return value is chosen during specialization.

```
bool_t xdrmem_putlong(XDR * xdrs, long * lp) {                          (1)
   if ((xdrs->x_handy -= sizeof(long)) < 0)                            (2)
         return (FALSE);                                               (3)
   *(long *)xdrs->x_private = (long)htonl((u_long)(*lp));              (4)
   xdrs->x_private += sizeof(long);                                    (5)
   return (TRUE);                                                      (6)
}                                                                      (7)
```

Fig. 13. The result of evaluation-time analysis of `xdrmem_putlong` (underlined constructs are dynamic, overlined constructs are static and dynamic).

2.1.2. Evaluation-time analysis

The result of binding-time analysis indicates the expressions that depend only on the static information and thus can be evaluated during specialization. This information is not, however, sufficient to ensure correct specialization. Problems may arise when certain kinds of static values are to be encoded in the residual program and when a variable assigned a static value is subsequently considered to be dynamic. We begin by examining these issues in more detail, and then present the analysis that addresses them.

When a static expression occurs in a dynamic context, the value of this expression must be residualized. Nevertheless, some values are *non-liftable*, i.e. they cannot be meaningfully represented in the specialized code. A pointer to a local variable is always non-liftable. When specialization is carried out at compile time, pointers to global variables are also non-liftable. Floating-point numbers may be considered non-liftable, because of the difference between the precision of the textual and internal representations. In the rest of this section, we consider all pointers to be non-liftable.

The flow-sensitive binding-time analysis of Tempo implies that a variable can be considered to be static at some occurrences and dynamic at others. For example, a variable that is assigned a static value in a conditional with a dynamic test is considered dynamic at any reference after the conditional, even though there is no intervening dynamic (i.e. residualized) assignment. This case is illustrated by Fig. 7(a), if the test expression is dynamic. A similar situation arises when a global variable that is assigned a static value is referenced by some code that is outside the specialized fragment, implying that the global variable is live at the end of the specialized fragment. In these two situations, the static assignment must be treated as both static and dynamic to ensure that the dynamic reference or the external reference is meaningful in the residual program.

Evaluation-time analysis addresses these issues [26]. The analysis reannotates every non-liftable static expression that occurs in a dynamic context as dynamic. This reannotation of an expression may in turn cause non-liftable subexpressions to occur in a dynamic context, and thus provoke a sequence of such reannotations. The analysis also maintains a set of the variables for which there is a subsequent dynamic reference but no reaching dynamic initialization along the current control-flow path. Any static assignment reaching such a reference is reclassified as both static and dynamic.

Fig. 13 shows the result of evaluation-time analysis of `xdrmem_putlong`. The expression `*(long *)xdrs->x_private` on the left-hand side of the assignment in line (4) is annotated as static in the result of the binding-time analysis (Fig. 12) but occurs

```
bool_t xdrmem_putlong((XDR * xdrs)^EV&RES, (long * lp)^ID) {^REB                (1)
  (if ((xdrs->x_handy -= sizeof(long)) < 0)                                    (2)
        return (FALSE);)^EV                                                    (3)
  (*(long *)xdrs->x_private = (long)htonl((u_long)(*lp));)^ID                   (4)
  (xdrs->x_private += sizeof(long);)^EV                                        (5)
  (return (TRUE);)^EV                                                          (6)
}^REB                                                                          (7)
```

Fig. 14. The result of action analysis of `xdrmem_putlong`.

in a dynamic context, because the right-hand side of the assignment is dynamic. The value of the left-hand side of an assignment is always an address, and thus a non-liftable value. The evaluation-time analysis reannotates this expression as dynamic. Its subexpression `xdrs`, which is a pointer, must then be considered dynamic as well, as shown in Fig. 13. Following this adjustment, there is a dynamic reference to `xdrs`, but no corresponding dynamic initialization. The only reaching initialization is the first parameter of `xdrmem_putlong`. This parameter becomes static and dynamic, implying that the corresponding argument becomes static and dynamic at each call site. Finally, if further values are to be marshaled, the pointer `xdrs->x_private` is needed after returning from `xdrmem_putlong`. In this case, the incrementing of this pointer in line (5) is also reannotated as static and dynamic.

2.1.3. Action analysis

The final analysis of Tempo is action analysis. This analysis determines how each construct should be specialized based on the evaluation times of its subterms. Action analysis is not essential, but simplifies the construction of a dedicated specializer to perform compile-time or run-time program specialization, or data specialization.

Tempo uses five actions: evaluate (EV), reduce (RED), rebuild (REB), identity (ID), and evaluate/residualize (EV & RES). Most of these actions are illustrated by the result of action analysis of the `xdrmem_putlong` function (Fig. 14). The parameter `xdrs` that is annotated as static and dynamic in Fig. 13 is annotated as evaluate/residualize here. The parameter is thus bound at specialization time, but also appears in the residual program. The parameter `lp` that is annotated as dynamic in Fig. 13 is annotated as identity here and thus only appears in the residual program. The body of the function is annotated as rebuild (indicated by the annotations on the opening and closing braces); this block must be reconstructed because it contains some dynamic computations. The statements in lines (2), (3), (5), and (6) are annotated as completely static in the result of the evaluation-time analysis. The action analysis annotates these statements EV, meaning that they are evaluated away during specialization. Finally, the assignment in line (4) is annotated as completely dynamic in the result of the evaluation-time analysis. It is thus annotated as ID by the action analysis, meaning that it is reproduced unchanged in the specialized program.

The treatment of `xdrmem_putlong` does not illustrate the reduce action. This action would be used if the code following the conditional were instead moved to the else branch of the conditional. In that case, it would be possible to reduce the conditional

during specialization, but not necessarily to completely evaluate it, because one of the branches would contain some code that should be residualized.

2.2. Specializer generation and specialization phases

For each form of specialization, Tempo creates a dedicated specializer, also known as a *generating extension* [19,29], based on the action-annotated program. This dedicated specializer is compiled and provided to users. If the dedicated specializer was created for compile-time specialization, a user links the specializer with any needed external functions and applies the resulting program to a set of configuration values provided in an auxiliary *specialization context file* to obtain specialized C source code. If the dedicated specializer was created for run-time specialization, a user links the specializer with the program that should use the specialized code, applies the specializer to the configuration values at the point in the program execution when they become available, and then invokes the generated code directly, as needed. Data specialization is similarly carried out at run time. We now examine each form of specialization in more detail.

2.2.1. Compile-time specialization

For compile-time specialization, the dedicated specializer consists of two kinds of functions: *code-generation functions* that construct the specialized program, and *evaluation functions* that contain the fragments annotated as EV by the action analysis. The essence of the dedicated specializer for `xdrmem_putlong` is shown in Fig. 15. Tempo constructs the dedicated specializer such that source program variables are represented as fields in the global structures `_local_sstore` and `_store`. The specializer initializes the configuration parameter `xdrs` of `xdrmem_putlong` by setting the location `_local_sstore.xdrmem_putlong_xdrs` to the value indicated by the user. Subsequently, the actual specialization process begins by invoking the function `specialize_putlong`. In this simplified definition, this function calls `instantiate` to generate specialized code. The function `instantiate` is applied to a string, representing the code to be generated, and some function pointers, which are invoked in sequence to instantiate the string positions indicated by `%s` with the results of specializing the subterms. In the example, there are three function pointers: `RES_1` for the EV code in lines (2) and (3) of the action-annotated function `xdrmem_putlong` (Fig. 14), `RES_2` for the ID assignment in line (4) of the action-annotated function, and `RES_3` for the EV code in lines (5) and (6) of the action-annotated function. The functions `RES_1` and `RES_3` call evaluation functions `EV_1` and `EV_2`, respectively, which execute the original code. The function `RES_2` simply returns the dynamic assignment statement.

After specialization, Tempo provides an optional post-processing phase that performs inlining, resugaring of constructs eliminated during the initial parsing phase (Fig. 5), and some simplifications of the specialized code.

2.2.2. Run-time specialization

Unlike compile-time specialization, where the specialized code is subsequently compiled using a standard compiler, run-time specialization must generate executable code directly from the dynamic source-program constructs. To address this issue, Tempo collects

```
// Specialization state
struct _local_sstore_s {
  struct XDR *xdrmem_putlong_xdrs;
  ...
} _local_sstore;

struct _global_sstore_s {
  int xdrmem_putlong_return;
  ...
} _store;

// Code generation functions
char *specialize_putlong() {
  return
    instantiate(
      "void xdrmem_putlong(XDR * xdrs, long * lp) { %s %s %s }",
      RES_1, RES_2, RES_3);
}

char *RES_1() {
  EV_1();
  return "{}";
}

char *RES_2() {
  return "*(long *)xdrs->x_private = (long)htonl((u_long)(*lp));";
}

char *RES_3() {
  EV_2();
  return "{}";
}

// Evaluation functions
void EV_1() {
  if ((_local_sstore.xdrmem_putlong_xdrs->x_handy -= sizeof(long)) < 0) {
    _store.xdrmem_putlong_return = FALSE;
    return;
  }
}

void EV_2() {
  _local_sstore.xdrmem_putlong_xdrs->x_private += sizeof(long);
  _store.xdrmem_putlong_return = TRUE;
  return;
}
```

Fig. 15. The dedicated compile-time specializer for `xdrmem_putlong`, as generated by Tempo.

the dynamic (REB or ID) constructs into a *template file*, which is compiled in advance using `gcc` and used at run time as a repository of fragments of executable code, known as *templates* [31], from which to construct a specialized definition. Fig. 16 shows the function corresponding to `xdrmem_putlong` in the template file. Because the only dynamic code in `xdrmem_putlong` is a single dynamic assignment, this function contains only one template. In general, a function can contain multiple templates, each corresponding to a sequence of REB or ID constructs and delimited by labels in the template file.

The actual specialization process is carried out by a dedicated run-time specializer that evaluates the static constructs, copies selected templates into a buffer representing the specialized definition, and instantiates these templates with computed static values and with appropriate branch offsets. These operations are simple and thus the cost of

```
extern void tmp_xdrmem_putlong(struct XDR *xdrs, int *lp) {
  // Bookkeeping data for template management
  static void *LA_tmp_xdrmem_putlong[2] =
   (void *)tmp_xdrmem_putlong,
    (void *)_tmp_xdrmem_putlong;

  // ID construct from the source program
  *(long *)xdrs->x_private = (long)htonl((u_long)(*lp));
}
```

Fig. 16. The template file function for `xdrmem_putlong`.

```
void *rts_xdrmem_putlong(struct XDR *xdrs) {
  // Allocate space for the specialized definition and for any large residualized values
  char *buffer = (*rts_alloc_code)(20000);                                          (1)
  char *data_buffer = (*rts_alloc_data)(20000);
  char *code_ptr = buffer;
  char *data_ptr = data_buffer;
  char *spec_ptr = code_ptr;
  char *temp_43_addr;

  temp_43_addr = 0;

  // Copy a template into the buffer reserved for the specialized definition
  DUMP_TEMPLATE(code_ptr, temp_43_addr, (char *)tmp_xdrmem_putlong + 0, 48);        (2)

  // Instantiate the specialized definition with the offset of a library function
  PATCH_LIB_CALL(temp_43_addr, (char *)tmp_xdrmem_putlong + 0, 20);                 (3)

  // Perform the remaining static computations
  if ((xdrs->x_handy -= sizeof(long)) < 0) {
    _xdrmem_putlong_return = FALSE;

    // Return a pointer to the specialized definition
    return (void *)spec_ptr;
  }
  xdrs->x_private += sizeof(long);
  _xdrmem_putlong_return = TRUE;

  // Return a pointer to the specialized definition
  return (void *)spec_ptr;
}
```

Fig. 17. Dedicated run-time specializer for `xdrmem_putlong`.

specialization is typically amortized after only a few uses of the generated code [59]. The specializer for `xdrmem_putlong` is shown in Fig. 17. This function allocates space for the specialized instance (line (1)), emits the only template associated with the function (line (2)), instantiates a library call (the call to `htonl`) in this template with the offset of the called function from the position of the call in the specialized code (line (3)), and then performs the static computations. In general, templates are emitted throughout the run-time specializer, at points corresponding to the positions of the dynamic code in the source program. The result of the run-time specializer is a pointer to the specialized function. If the corresponding source function returns a static result, this value is returned in a global variable, here `_xdrmem_putlong_return`.

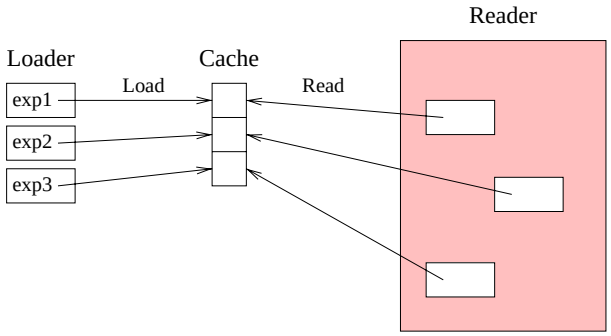

Fig. 18. Data specialization (exp1, exp2, and exp3 are static subexpressions of the source program).

Consel and Noël [14] and Noël et al. [59] present more details about the run-time specializer in Tempo, from both a theoretical and a practical perspective. Refinements of this approach allow inlining of specialized functions within the residual program [37].

2.2.3. Data specialization

Data specialization separates the computation of the program into a *loader* and a *reader*. The loader is invoked once with the static configuration values and evaluates the static constructs of the source program. This phase stores (i.e. loads) the results of evaluating the static constructs in a data structure known as a *cache*, which amounts to the specialized data. The reader is invoked as needed with the dynamic inputs and carries out the remaining computations. When the reader needs the value of a static subexpression, it retrieves this value from the cache. The relationship between the loader, the reader, and the cache is illustrated in Fig. 18.

Because both the loader and reader are generated at compile time, before the static data is known, data specialization does not simplify conditionals or unroll loops. The lack of these transformations implies that data specialization often gives less performance improvement than program specialization, in which new code is generated based on the static values. On the other hand, because data specialization cannot lead to code explosion, it is useful for some applications where excessive code would be generated by program specialization [7].

Fig. 19 shows the loader and reader generated for `xdrmem_putlong`. We have slightly modified the definition of this function to place the code following the conditional statement in the original implementation (Fig. 2) in the else branch of the conditional; cache management in the current implementation of data specialization in Tempo requires that if any branch contains a return statement then they all do. Although static and dynamic portions of `xdrmem_putlong` are largely disjoint, because static conditionals are not reduced by data specialization, the static value of the conditional test (line (2) of Fig. 14) must be communicated from the loader to the reader via the cache. Lines (1) and (2) in the loader initialize the cache entry according to the value of the test. Line (3) in the reader accesses this value to choose the branch to execute, without re-evaluating the test expression. Both the loader and the reader return a pointer to the next free position in

```
void **xdrmem_putlong_2_loader(struct XDR *xdrs, void **Cache) {
  if ((xdrs->x_handy -= sizeof(long)) < 0) {
    *((int *)Cache) = 1;                          // Test expression value      (1)
    _xdrmem_putlong_return = FALSE;
    return Cache + 1;                             // New cache pointer
  }
  else {
    *((int *)Cache) = 0;                          // Test expression value      (2)
    xdrs->x_private += sizeof(long);
    _xdrmem_putlong_return = TRUE;
    return Cache + 1;                             // New cache pointer
  }
}

void **xdrmem_putlong_2_reader(struct XDR *xdrs, int *lp, void **Cache) {
  if (*((int *)Cache))                            // Access cached test value   (3)
    return Cache + 1;
  else {
    *(long *)xdrs->x_private = (long)htonl((u_long)(*lp));
    return Cache + 1;
  }
}
```

Fig. 19. The loader and reader for data specialization of `xdrmem_putlong`.

the cache. Because passing a value from the loader to the reader through the cache involves memory references, the cost of using the cache may be higher than the cost of simply re-evaluating a very simple expression, such as a variable reference. Tempo thus gives the program developer some control over the caching strategy.

Because data specialization has slightly different properties than program specialization, the analysis phase of Tempo must be directed specifically to prepare the code for data specialization. Nevertheless, the differences are minor, and the same analysis engine is used. More information about data specialization in Tempo is available in the work of Chirokoff et al. [7] and Lawall [37].

2.3. Assessment

The research on the engines used for program specialization has paralleled and built upon developments in optimizing compilation. As a result, most of the work has been directed towards increasing the accuracy of program analyses and introducing ever more sophisticated transformations. The goal of this work was to widen the scope of the programs that would successfully specialize. Nevertheless, these efforts were pursued in general and did not target programs in a particular domain area.

In contrast, our starting point in the development of Tempo was to study programs in a particular domain area, namely operating systems. This study permitted us to precisely identify situations that required both the use of existing analysis techniques (e.g., flow sensitivity) and the development of new techniques (e.g., return sensitivity). Specialization techniques were introduced in Tempo, not a priori, but as we explored application domains. For example, run-time specialization was motivated by the need to specialize systems code with respect to run-time values, as suggested by operating system researchers. Data specialization was introduced in the course of studying the application of Tempo to scientific programs. In our use of Tempo, we have found that the analyses and

specialization strategies motivated by this domain analysis give good results for a variety of applications including interpreters and systems code.

3. Applications

Tempo has mainly been used in two areas: operating systems and compiler generation. In this section, we give an overview of applications of Tempo in these areas and discuss the resulting performance improvements. Tempo has additionally been applied to a variety of scientific algorithms [36,59]. Beyond the specialization of specific applications, Tempo has also served as a *back-end* specializer for C++ and Java programs, by first translating such programs into an intermediate representation expressed using the C language.

3.1. Operating systems

An early success of the Tempo project was the specialization of the XDR library of the RPC mechanism. The computations optimized away by Tempo include the testing of the coding direction, the checking for buffer overflow, and the testing of the return status. The last optimization critically relies on return sensitivity, allowing a static return value in a callee to flow back to a caller. This feature allows computations depending on the return value to be performed during specialization. We specialized the XDR layer with respect to different values for the size of an array to be used as an RPC argument [53,55]. The specialized versions of this layer ran between 165% and 235% faster than the original version on a Pentium under Linux. The overall round-trip performance improvement of the RPC was 35% on the same platform. These experiments were carried out using compile-time specialization. Kono and Masuda have applied run-time specialization using Tempo to the problem of marshaling data and obtained similar performance improvements [35].

Another systems application studied in the context of Tempo was UNIX signals. This mechanism allows processes to communicate events. Specialization opportunities occur when a process sends a given signal multiple times to the same destination process in the course of some collaborative work. In this situation a number of comparisons and conditionals can be eliminated; the residual code solely consists of operations to deliver the signal to the target process. When specialized, the resulting system call is 65% faster than the original code [53].

3.2. Compiler generation

The long line of work on specializing interpreters, initiated by Jones, has found a number of follow-ups in our operating systems work. One such application is the specialization of a low-level interpreter for selecting packets from a network interface, named the Berkeley packet filter (BPF) [52]. This kernel-resident interpreter runs packet filter programs written in a byte code by application programmers. The BPF, like any interpreter, can be specialized with respect to a particular program, thus compiling away most, if not all, of the interpretation overhead. The BPF interpreter was specialized both at compile time and run time to account for opportunities existing at these two stages. On a Pentium, the compile-time specialized BPF interpreter was 3.4 times faster than the

original version, and 1.95 times faster when specialized at run time [73]. The difference in performance is due to the fact that compile-time specialization produces a program that can be globally optimized by a conventional compiler. In contrast, the run-time specializer assembles binary templates that are only locally optimized.

Another case study is an interpreter for a language, named PLAN-P, that allows a programmer to define application-specific network protocols [74,75]. To achieve maximum flexibility, protocols need to be deployed dynamically over a network of programmable routers. In order to satisfy portability, safety, and security constraints, protocols need to be deployed as source code, making interpretation a natural execution strategy. Nevertheless, interpretation is not efficient; some form of compilation is needed. Specializing the PLAN-P interpreter with respect to a PLAN-P protocol has the effect of compilation. By performing this specialization using Tempo's run-time specializer, we obtain executable code directly, thus essentially achieving just-in-time compilation [1]. The specialized PLAN-P interpreter was found to be 35% faster than an equivalent compiled and optimized Java program [73], thus showing that this approach successfully reconciles the need to manipulate the source program for portability and verification purposes with the need for efficiency.

Outside of the realm of languages directed toward operating systems applications, we have also considered specialization of interpreters for Java byte code and OCaml byte code [73]. Compile-time specialization of a handwritten Java byte code interpreter gave speed-ups of 3–4 times on standard benchmarks, and run-time specialization gave speed-ups of 25–94%. While the compile-time specialized code was as much as 3 times faster than existing interpreters, specialization was not able to achieve the degree of optimization provided by offline or JIT compilers, which were 2–10 times faster. This experiment thus illustrates the limits of compilation via specialization, which essentially only inlines the implementation of each construct, as compared to hand-crafted optimizing compilers, which typically perform multiple kinds of optimizations. Similar results were obtained in the case of the OCaml 1.05 interpreter [43].

3.3. Tempo as a back-end specializer

Specialization has been explored for a variety of realistic languages. Nevertheless, developing a specializer for such a language remains a daunting task. A promising alternative is to explore specialization for a new language by translating programs into a language for which there already exists a specializer. This approach exploits an existing and stable infrastructure, and thus permits the developer to concentrate on issues particular to the new language. In our exploration of this approach using Tempo, we have found that the C language is suitable for accurately expressing the semantics of a wide variety of programming languages. Indeed, a number of compilers use C as their target language [17,56,65]. Our study of back-end specialization mainly focused on two languages whose specialization had not previously been addressed, namely C++ and Java.

To translate C++ programs into C, we use the `cfront` compiler. The translated code is amenable to successful specialization, modulo minor transformations to reduce its size by eliminating unnecessary program fragments [76]. Compile-time specialization of C++ was used to specialize industrial-strength systems code: parts of the Chorus inter-process

communication subsystem. The main limitation of this approach was that it produced C code rather than C++ code, thus making it difficult to understand the residual code and complicating the integration of the residual code into Chorus.

This limitation was addressed by a subsequent project that used Tempo as a back-end specializer for Java. The translation from Java to C was performed by the Harissa compiler, developed in the Compose group [56,57]. Numerous optimizations were built into the Harissa compiler, including method inlining driven by a class hierarchy analysis. Some extensions were made to Harissa to generate auxiliary declarations used by Tempo, such as the analysis context file, and to facilitate the translation back to Java after specialization [67]. The translation from Java to C exposed specialization opportunities concerning language features such as virtual calls, casts, and array references. Yet, some specialization opportunities inherent in the source program appeared to be difficult for Tempo to exploit because of the heavy use of data structures and dynamic memory allocation in the code generated by Harissa. To cope with this kind of code, Tempo's analyses were extended with the polyvariant treatment of structures described in Section 2. To make back-end specialization transparent to the programmer, a translator from C to Java was also developed. This back translator only handles the subset of C (and specific program patterns) that can be generated by Harissa and output by Tempo. More information about the specialization of Java can be found in the work of Schultz et al. describing the resulting specializer, JSpec [68–70].

3.4. Assessment

These experiments show that specialization is effective for programs written using a generic, layered, or interpretive structure. In these cases, it provides aggressive customization with respect to configuration values. Nevertheless, specialization is not a cure-all, as it does not perform general optimizations such as those that reorganize the program (e.g., loop scheduling). Indeed, as demonstrated by Schultz et al., specializers and optimizing compilers are complementary [69].

Existing work has focused on specializing specific applications. Another direction consists of exploring the combination of program specialization and application generators. To ease their development, application generators usually produce generic code and rely on highly parametrized libraries. Because both the libraries and the application generator are fixed, all possible generated applications can benefit from a fixed set of specialization contexts. Encouraging results along this line have been obtained in the setting of the RPC, which is implemented as a combination of a stub compiler and the XDR library.

4. Towards improving the usability of Tempo

We have found that the analyses described in Section 2 have reached a satisfactory level of precision. Therefore, our main concern has recently turned to improving the usability of Tempo. To cope with both a realistic language such as C and realistic applications, existing program specializers require the program developer to be proficient in using a number of complex parameters and mechanisms to finely configure the specialization process. Even when properly configured, a program specializer may not produce a residual

program that uses configuration values according to the programmer's expectations. Indeed, programs are often either under-specialized or over-specialized. In our experience, the unpredictability of the specialization process greatly increases the effort and time required to obtain the desired degree of specialization.

In this section, we present our first results towards improving the usability of Tempo: a declarative language that allows a program developer to describe specialization intentions. Declarations provided using this language enable automatic configuration of Tempo and are verified to be respected by the analysis phase. This verification is critical for making the specialization process predictable with respect to the declarations. To give further assistance, some visualization tools have been developed, which are also described here.

4.1. Declaring specialization

Successful specialization typically requires that the program developer have an intuitive understanding of how static information should propagate through the program. The declaration language provided by Tempo enables the program developer to describe specialization opportunities as a set of *specialization scenarios* that are written in auxiliary specialization modules while the source code is being developed [39–41]. A specialization scenario identifies a function, global variable, or data structure that is of interest for specialization, and declares the binding-time context, i.e. static or dynamic, in which specialization involving this construct should be carried out. To create such a scenario, the developer copies the C type declaration of the affected construct into the specialization module, and annotates this declaration with binding-time information. A specialization scenario thus amounts to an extended type declaration, in which the extra information describes the developer's intentions, not as regards the values being manipulated, but as regards the time at which they become available.

We illustrate the process of creating specialization scenarios in the context of the specialization of the XDR library described in Sections 1 and 2. The scenarios are organized into the modules `xdr` and `xdrmem` following the layer decomposition of the library. The `xdrmen` layer defines the function `xdrmem_putlong` (Fig. 2) that writes the marshaled value into the output buffer. When the position in the output buffer (as indicated by `xdrs->x_handy` and `xdrs->x_private`) is known, the buffer overflow check and pointer increment performed by this function can be carried out at specialization time, leaving only the copying of the data into the buffer in the specialized code. The scenarios are derived from the C declarations shown in Fig. 20, corresponding to the data structure and the function that are involved in this specialization.

Figs. 21 and 22 show specialization scenarios that describe the desired specialization opportunities. The scenario `Btxdrmem_putlong`, given in the module `xdrmem` (Fig. 21), specifies that the function `xdrmem_putlong` is defined in file `xdr_mem.c` and can be specialized if its arguments have the following properties: the argument `xdrs` is a static pointer, the result of dereferencing this pointer satisfies the scenario `BtXDR`, and the argument `lp` is completely dynamic. The scenario `BtXDR` is imported from the module `xdr` (Fig. 22) and amounts to the C declaration of the `XDR` structure annotated to indicate that the fields `x_handy` and `x_private` are static. The scenario `Btxdrmem_putlong` additionally contains a `needs` clause specifying that invocation of `htonl` within the

```
in file xdr.h:

 struct XDR { ...
              char * x_private;
              ...
              int x_handy;
            };

in file xdr_mem.c:

 bool_t xdrmem_putlong ( struct XDR * xdrs, long * lp );
```

Fig. 20. C declarations for the `xdr` and `xdrmem` layers.

```
in file xdr_mem.mdl:

 Module xdrmem {
  Imports { From xdr.mdl { BtXDR; }                  // Import scenarios from other modules
            From library.mdl { Bthonl; }
  }
  Defines {
    From xdr_mem.c {
      Btxdrmem_putlong :: intern xdrmem_putlong (    // Define a scenario for xdrmem_putlong
                    BtXDR(struct XDR) S(*) xdrs,     // Static pointer to a structure XDR
                                                     //  that satisfies the scenario BtXDR
                    D(long *) lp )                   // Dynamic pointer to dynamic long
               { needs { Bthtonl; } };               // List scenarios for called functions
      ... }
    ...}
  Exports { Btxdrmem_putlong; ... }}                 // Export scenarios declared locally
```

Fig. 21. The specialization module for the `xdrmem` layer.

body of the `xdrmem_putlong` should satisfy the imported scenario `Bthtonl`. In the result of specialization previously shown in Fig. 3, the body of `xdr_int_spe` amounts to a specialized instance of `xdrmem_putlong`. According to the specialization scenario `Btxdrmem_putlong`, specialization has removed the overflow check and the pointer increment performed by `xdrmem_putlong`, leaving only the buffer copy.

This example illustrates some aspects of the use of specialization scenarios. Even though creating specialization scenarios increases the amount of information that has to be provided by the program developer, these declarations are proportional in size to the C type declarations of the relevant constructs, and are thus typically much smaller than the entire source program. Furthermore, collecting scenarios into specialization modules centralizes specialization information related to a particular functionality and facilitates the understanding and reuse of specialization scenarios.

4.2. Verifying specialization

Specialization scenarios are checked during the binding-time analysis. Tempo regards a specialization scenario as a contract, expressing the developer's exact intentions as to the desired result of specialization. Consequently, if the analysis cannot satisfy a binding-time property declared in a scenario, Tempo aborts the specialization process with an error

in file xdr.mdl:

```
Module xdr {
 Imports { ... }                                    // Import scenarios from other modules
 Defines {
   From xdr.h {
    BtXDR :: struct XDR { ...                       // Define a scenario for structure XDR
                          D(char) S(*) x_private;   // Static pointer to a dynamic char
                          ...
                          S(int) x_handy;           // Static int
                        };
    ... }
   ... }
 Exports { BtXDR; ...}}                             // Export scenarios declared locally
```

Fig. 22. The specialization module for the `xdr` layer.

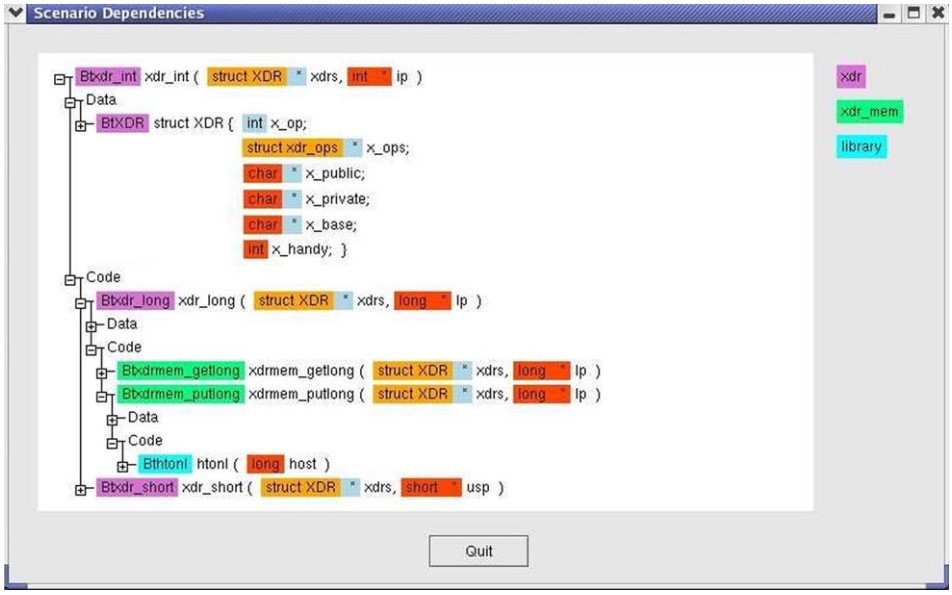

Fig. 23. The specialization scenario dependency graph for the specialization of `xdr_int`.

message describing the binding-time mismatch rather than producing an under-specialized program. Similarly, if the analysis determines that some data is available at specialization time but the developer has declared that this data should not be used, the analysis considers the data to be dynamic, thus following the developer's intentions rather than producing an over-specialized program. Overall, if Tempo produces a specialized program, this program has exactly the form that the developer expects. Thus, the specialization process is predictable.

Once a function and all of its callees have been implemented along with the corresponding specialization scenarios, the feasibility of the interactions between the

scenarios can be checked using Tempo's binding-time analysis. Developing and checking scenarios incrementally thus facilitates the treatment of large programs.

4.3. Visualizing specialization

Tempo provides some support tools to assist the developer in making code specializable. Prior to preprocessing, graphical tools allow the developer to visualize the specialization declarations. One tool shows the hierarchy of the specialization modules involved in a given specialization and another allows inspection of the dependencies between the scenarios. To illustrate the use of the latter tool, Fig. 23 shows the scenario dependencies starting from the scenario `Btxdr_int`. For each scenario, referenced scenarios are classified as "Data" if they are associated with data structures or global variables and as "Code" if they refer to functions. To enable the developer to easily determine the module in which a given scenario is defined, the name of a module and the names of the scenarios that it contains are shown in the same color. Within each scenario, types are colored to reflect binding-time properties. The colors and the connections in the dependency graph help the developer track the propagation of static parameters throughout the code before initiating Tempo's preprocessing phase. After preprocessing, the developer can check the intermediate results of the different analyses by inspecting the annotated source files that are generated during this phase.

4.4. Assessment

Even though it is difficult to assess the improvement in the usability of a tool, based on our own experiments and the reports of others, we have observed that specialization modules have facilitated the use of Tempo by non-experts. Prior to the development of the declaration language, we had used Tempo both in teaching and in collaboration with industry. Nevertheless, these experiments, in particular the work with industry, were ultimately unsuccessful, because of the difficulty of obtaining the expected specialized program. Since the development of the declaration language, Tempo has been used in a graduate course on program specialization, in graduate student projects, and in an industrial project. In the graduate course, students were able to successfully specialize classical program specialization examples in less time than the students of the previous years. Other students have obtained the desired result from specialization of more complex programs, either independently or after only a short presentation of the language features. As a more realistic test, our declarative approach has also been used by an engineer at Philips to customize industrial code related to digital television.

These results are encouraging, but much work remains to enable the integration of Tempo in an industrial-strength software development process. Before embarking on specialization of an application, it could be useful to have an estimate of the expected improvement in terms of program size and execution time. An interactive debugger could help the program developer better understand why a given specialization scenario cannot be satisfied. It could also be beneficial to control aspects of the specialization process other than binding times, such as bounding the unrolling of specific loops. Finally, it could be useful to constrain the overall properties of the specialized program such as its maximum possible size.

5. Related work

Many of the innovations in Tempo as compared to existing program specialization technology are in the treatment of imperative constructs and programming styles found in C programs. Thus, we concentrate on other specializers that have been developed for this language. Other specializers for C programs have focused on only one kind of specialization, either compile-time program specialization, run-time program specialization, or data specialization.

5.1. Compile-time program specialization

C-Mix is a compile-time specializer for C programs [46]. The binding-time analysis of C-Mix is more efficient but less precise than that of Tempo. In particular, it is neither flow-sensitive nor context-sensitive, and does not distinguish the binding time of a pointer from that of the referenced value. The latter constraint eliminates the possibility of specialization-time calculations on pointers to dynamic values as used in the XDR example (Fig. 14). C-Mix provides *continuation-based partial evaluation* [4,10], in which the context of a conditional statement is propagated into each branch of the conditional where it is individually specialized with respect to information available in the branch. This strategy can increase the number of constructs that can be considered static, but can lead to code explosion [28]. Accordingly, Tempo does not provide this feature; the program developer can usually obtain the same effect in a controlled way by manual program transformation [53,73].

The designers of C-Mix have also considered the problem of making the specialization process more accessible to program developers. C-Mix provides a notation for specifying binding times for individual variables and an interactive interface that allows the program developer to determine the chain of inferences that lead to particular binding-time annotations [8].

5.2. Run-time program specialization

DyC is a run-time specializer for C programs [21]. The design of this system focuses on control of the specialization process and on optimizing the generated code. Source programs can be annotated to indicate where particular variables should become static or dynamic and to control the caching strategy for specialized code fragments. Nevertheless, the binding-time analysis of DyC is much less precise than that of Tempo. DyC's analysis treats scalar local variables but not global variables, pointers, or data structures, which are either always considered to be dynamic or require programmer annotations. Like Tempo, DyC organizes fragments of dynamic code into templates, but these are compiled using a special-purpose variant of a research compiler (Multiflow [45]) rather than on an existing standard compiler as in the case of Tempo. The result of template compilation is a sequence of code-emitting instructions that may perform optimizations based on static values. These optimizations cause DyC to generate more efficient code than Tempo on some examples [22]. Nevertheless, Tempo achieves a much greater level of portability due to the use of a standard widely available compiler (`gcc`) and a lower run-time specialization

cost due to the generation of the code associated with each template using a single memory copy.

The Fabius program specializer, targeted towards a pure, first-order subset of the functional language SML, also performs run-time specialization [38,42]. Like DyC, Fabius includes a dedicated code generator that emits residual machine instructions one by one and performs some run-time optimization of this code.

5.3. Data specialization

The implementation of data specialization in Tempo was partially inspired by the work by Knoblock and Ruf on data specialization for C programs [34]. Their system also separates code into a loader and a reader according to the results of a dependency analysis. They propose some optimizations particular to data specialization that reduce the size of the cached data. Interestingly, their dependency analysis includes features similar to Tempo's evaluation-time analysis, that we developed previously in Tempo for the needs of compile-time specialization. Nevertheless, their implementation is much less developed than that of Tempo, as it considers only a single function, does not treat pointers, and considers all code under dynamic control to be dynamic. An approach that is very similar to that of Knoblock and Ruf has also been explored in the context of Java [60].

6. Conclusions and future work

The goal of the Tempo project has been to make specialization a practical tool for optimizing realistic programs. Examination of realistic specialization cases prompted us to develop new analyses and transformation techniques for Tempo. The resulting specializer has been continuously tested against a variety of applications in operating systems, networking, and interpreters. The contributions of this research project reflect the breadth of its scope; they include results in program analyses, program transformation, language design, software engineering, operating systems, and networking.

Tempo's capabilities and features open a host of new research directions, particularly in the area of improving the accessibility of the technology. In the area of applications, as noted in Section 3.4, the domain of application generators is a promising target for specialization. In this case, codifying specialization opportunities exhibited by the libraries used by such generators allows all generated applications to easily benefit from specialization. In the area of general usability, as noted in Section 4.4, we are considering extensions to the expressiveness of the declaration language and the further development of associated tools. Finally, we would like to couple specialization with declarations and mechanisms to enable specialized code to be integrated safely and correctly as the program executes. This is particularly interesting for long-running systems that critically rely on performance but whose execution cannot be suspended.

Acknowledgements

Many people have contributed to the development of Tempo: Sandrine Chirokoff, Luke Hornof, Renaud Marlet, Gilles Muller, François Noël, Jacques Noyé, Scott Thibault,

Ulrik Pagh Schultz, and Nic Volanski. The authors would also like to thank the anonymous referees and Ralf Laemmel for their helpful comments on previous drafts of this paper.

Additional information and articles about Tempo can be found at http://compose.labri.fr.

References

[1] J. Aycock, A brief history of just-in-time, ACM Computing Surveys 35 (2) (2003) 97–113.
[2] G.J. Barzdins, M.A. Bulyonkov, Mixed computation and translation: linearisation and decomposition of compilers, preprint 791 from Computing Centre of Sibirian Division of USSR Academy of Sciences, Novosibirsk, 1988, p. 32.
[3] A. Birrell, B. Nelson, Implementing remote procedure calls, ACM Transactions on Computer Systems 2 (1) (1984) 39–59.
[4] A. Bondorf, Improving binding times without explicit CPS-conversion, in: ACM Conference on Lisp and Functional Programming, ACM Press, San Francisco, CA, USA, 1992, pp. 1–10.
[5] A. Bondorf, J. Jørgensen, Efficient analyses for realistic off-line partial evaluation, Journal of Functional Programming 3 (3) (1993) 315–346.
[6] M. Braux, J. Noyé, Towards partial evaluating reflection in Java, in: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, Boston, MA, USA, 2000, pp. 2–11.
[7] S. Chirokoff, C. Consel, R. Marlet, Combining program and data specialization, Higher-Order and Symbolic Computation 12 (4) (1999) 309–335.
[8] C-Mix$_{/II}$ user and reference manual, 2000. http://www.diku.dk/research-groups/topps/activities/cmix/download/.
[9] C. Consel, A tour of Schism: a partial evaluation system for higher-order applicative languages, in: PEPM'93 [62], pp. 66–77.
[10] C. Consel, O. Danvy, For a better support of static data flow, in: J. Hughes (Ed.), Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, vol. 523, Springer-Verlag, Cambridge, MA, USA, 1991, pp. 496–519.
[11] C. Consel, O. Danvy, Tutorial notes on partial evaluation, in: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, Charleston, SC, USA, 1993, pp. 493–501.
[12] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, N. Volanschi, Tempo: specializing systems applications and beyond, ACM Computing Surveys, Symposium on Partial Evaluation 30 (3) (1998).
[13] C. Consel, L. Hornof, F. Noël, J. Noyé, E. Volanschi, A uniform approach for compile-time and run-time specialization, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation, International Seminar, Dagstuhl Castle, Lecture Notes in Computer Science, No. 1110, 1996, pp. 54–72.
[14] C. Consel, F. Noël, A general approach for run-time specialization and its application to C, in: Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, FL, USA, 1996, pp. 145–156.
[15] D. Dussart, E. Bevers, K. De Vlaminck, Polyvariant constructor specialisation, in: PEPM'95 [63], pp. 54–65.
[16] A. Erosa, L. Hendren, Taming control flow: A structured approach to eliminating goto statements, in: Proceedings of the IEEE 1994 International Conference on Computer Languages, Toulouse, France, 1994, pp. 229–240.
[17] S. Feldman, D. Gay, M. Maimone, N. Schryer, A Fortran to C converter, in: Computing Science Technical Report 149, AT & T Bell Laboratories, Murray Hill, NJ, March, 1995.
[18] J.P. Gallagher, Tutorial on specialisation of logic programs, in: PEPM'93 [62], pp. 88–98.
[19] R. Glück, J. Jørgensen, Efficient multi-level generating extensions for program specialization, in: M. Hermenegildo, S. Doaitse Swierstra (Eds.), Proceedings of the 7th International Symposium on Programming Language Implementation and Logic Programming, Utrecht, The Netherlands, Lecture Notes in Computer Science, vol. 982, 1995, pp. 259–278.
[20] R. Glück, J. Jørgensen, An automatic program generator for multi-level specialization, Lisp and Symbolic Computation 10 (1997) 113–158.

[21] B. Grant, M. Mock, M. Philipose, C. Chambers, S. Eggers, DyC: an expressive annotation-directed dynamic compiler for C, Theoretical Computer Science 248 (1–2) (2000) 147–199.
[22] B. Grant, M. Philipose, M. Mock, C. Chambers, S. Eggers, An evaluation of staged run-time optimizations in DyC, in: Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation, PLDI'99, Atlanta, GA, USA, 1999, pp. 293–304.
[23] T.J. Hickey, D.A. Smith, Toward the partial evaluation of CLP languages, in: PEPM'91 [61], pp. 43–51.
[24] L. Hornof, J. Noyé, Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity, in: ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, Amsterdam, The Netherlands, 1997, pp. 63–73.
[25] L. Hornof, J. Noyé, Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity, Theoretical Computer Science 248 (1–2) (2000) 3–27.
[26] L. Hornof, J. Noyé, C. Consel, Effective specialization of realistic programs via use sensitivity, in: P. Van Hentenryck (Ed.), Proceedings of the Fourth International Symposium on Static Analysis, SAS'97, Lecture Notes in Computer Science, vol. 1302, Springer-Verlag, Paris, France, 1997, pp. 293–314.
[27] D. Johnson, W. Zwaenepoel, The Peregrine high-performance RPC system, Software – Practice and Experience 23 (2) (1993) 201–221.
[28] N. Jones, What *not* to do when writing an interpreter for specialisation, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation, International Seminar, Dagstuhl Castle, Lecture Notes in Computer Science, No. 1110, 1996, pp. 216–237.
[29] N. Jones, C. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, International Series in Computer Science, Prentice-Hall, 1993.
[30] N. Jones, P. Sestoft, H. Søndergaard, An experiment in partial evaluation: the generation of a compiler generator, in: J.-P. Jouannaud (Ed.), Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 202, Springer-Verlag, 1985, pp. 124–140.
[31] D. Keppel, S. Eggers, R. Henry, Evaluating runtime compiled value-specific optimizations, Technical Report 93-11-02, Department of Computer Science, University of Washington, Seattle, WA, 1993.
[32] S. Khoo, R. Sundaresh, Compiling inheritance using partial evaluation, in: PEPM'91 [61], pp. 211–222.
[33] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, R. Glück, Fortran program specialization, in: U. Meyer, G. Snelting (Eds.), Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen, Justus-Liebig-Universität, Giessen, Germany, 1994, pp. 45 –54, Report No. 9402.
[34] T. Knoblock, E. Ruf, Data specialization, in: PLDI'96 [64], 1996, pp. 215 –225, Also TR MSR-TR-96-04, Microsoft Research.
[35] K. Kono, T. Masuda, Efficient RMI: Dynamic specialization of object serialization, in: Proceedings of the 20th International Conference on Distributed Computing Systems, IEEE Computer Society Press, Taipei, Taiwan, 2000, pp. 308–315.
[36] J. Lawall, Faster Fourier transforms via automatic program specialization, in: J. Hatcliff, T.A. Mogensen, P. Thiemann (Eds.), Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School, Lecture Notes in Computer Science, vol. 1706, Springer-Verlag, Copenhagen, Denmark, 1998, pp. 338–355.
[37] J. Lawall, Implementing circularity using partial evaluation, in: O. Danvy, A. Filinski (Eds.), Programs as Data Objects, Second Symposium, PADO 2001, Springer, Aarhus, Denmark, Lecture Notes in Computer Science, vol. 2053, 2001, pp. 84–102.
[38] P. Lee, M. Leone, Optimizing ML with run-time code generation, in: PLDI'96 [64], pp. 137–148.
[39] A.-F. Le Meur, Approche déclarative à la spécialisation de programmes C, Thèse de doctorat, Université de Rennes 1, France, 2002.
[40] A.-F. Le Meur, C. Consel, B. Escrig, An environment for building customizable software components, in: IFIP/ACM Conference on Component Deployment, Berlin, Germany, 2002, pp. 1–14.
[41] A.-F. Le Meur, J. Lawall, C. Consel, Specialization scenarios: a pragmatic approach to declaring program specialization, Higher-Order and Symbolic Computation 17 (1) (2004) 47–92.
[42] M. Leone, P. Lee, Lightweight run-time code generation, in: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, FL, USA, Technical Report 94/9, University of Melbourne, Australia, 1994, pp. 97–106.
[43] X. Leroy, The Objective Caml System Release 1.05, 1997.

[44] B. Locanthi, Fast bitblt() with asm() and cpp, in: European UNIX Systems User Group Conference Proceedings, EUUG, AT & T Bell Laboratories, Murray Hill, 1987, pp. 243–259.
[45] P. Lowney, S. Freudenberger, T. Karzes, W.D. Lichtenstein, R. Nix, J. O'Donnell, J. Ruttenberg, The Multiflow trace scheduling compiler, The Journal of Supercomputing 7 (1–2) (1993) 51–142.
[46] H. Makholm, Specializing C – an introduction to the principles behind C-Mix$_{/II}$, Technical Report, Computer Science Department, University of Copenhagen, 1999.
[47] K. Malmkjær, Program and data specialization: Principles, applications, and self-application, Master's Thesis, DIKU University of Copenhagen, 1989.
[48] K. Malmkjær, P. Ørbæk, Polyvariant specialisation for higher-order, block-structured languages, in: PEPM'95 [63], pp. 66–76.
[49] M. Marinescu, B. Goldberg, Partial-evaluation techniques for concurrent programs, in: ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, Amsterdam, The Netherlands, 1997, pp. 47–62.
[50] R. Marlet, C. Consel, P. Boinot, Efficient incremental run-time specialization for free, in: Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation, PLDI'99, Atlanta, GA, USA, 1999, pp. 281–292.
[51] H. Massalin, C. Pu, Threads and input/output in the Synthesis kernel, in: Proceedings of the Twelfth Symposium on Operating Systems Principles, Arizona, 1989, pp. 191–201.
[52] S. McCanne, V. Jacobson, The BSD packet filter: a new architecture for user-level packet capture, in: Proceedings of the Winter 1993 USENIX Conference, USENIX, San Diego, CA, USA, 1993, pp. 259–269.
[53] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, C. Goel, C. Consel, G. Muller, R. Marlet, Specialization tools and techniques for systematic optimization of system software, ACM Transactions on Computer Systems 19 (2001) 217–251.
[54] U. Meyer, Techniques for partial evaluation of imperative languages, in: PEPM'91 [61], pp. 94–105.
[55] G. Muller, R. Marlet, E. Volanschi, C. Consel, C. Pu, A. Goel, Fast, optimized Sun RPC using automatic program specialization, in: Proceedings of the 18th International Conference on Distributed Computing Systems, IEEE Computer Society Press, Amsterdam, The Netherlands, 1998, pp. 240–249.
[56] G. Muller, B. Moura, F. Bellard, C. Consel, Harissa: a flexible and efficient Java environment mixing bytecode and compiled code, in: Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems, Usenix, Portland, OR, USA, 1997, pp. 1–20.
[57] G. Muller, U. Schultz, Harissa: a hybrid approach to Java execution, IEEE Software (1999) 44–51.
[58] G. Muller, E. Volanschi, R. Marlet, Scaling up partial evaluation for optimizing the Sun commercial RPC protocol, in: ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, Amsterdam, The Netherlands, 1997, pp. 116–125.
[59] F. Noël, L. Hornof, C. Consel, J. Lawall, Automatic, template-based run-time specialization: Implementation and experimental study, in: International Conference on Computer Languages, IEEE Computer Society Press, Chicago, IL, 1998, pp. 132 –142, Also available as IRISA Report PI-1065.
[60] J. Park, M.-S. Park, Using indexed data structures for program specialization, in: ACM SIGPLAN ASIA-PEPM 2002, Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, Aizu, Japan, 2002, pp. 61–69.
[61] Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, New Haven, Connecticut, USA, 1991.
[62] Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, Copenhagen, Denmark, 1993.
[63] Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, La Jolla, CA, USA, 1995.
[64] Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, 31(5), Philadelphia, PA, USA, 1996.
[65] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, S. Watterson, Toba: Java for applications – a way ahead of time (WAT) compiler, in: Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems, Usenix, Portland, OR, USA, 1997, pp. 41–53.
[66] M. Schroeder, M. Burrows, Performance of Firefly RPC, ACM Transactions on Computer Systems 8 (1) (1990) 1–17.

[67] U.P. Schultz, Object-oriented software engineering using partial evaluation, Ph.D. Thesis, Université de Rennes 1, France, 2000.
[68] U. Schultz, J. Lawall, C. Consel, Specialization patterns, in: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ASE 2000, IEEE Computer Society Press, Grenoble, France, 2000, pp. 197–208.
[69] U. Schultz, J. Lawall, C. Consel, Automatic program specialization for Java, ACM Transactions on Programming Languages and Systems 25 (4) (2003) 452–499.
[70] U. Schultz, J. Lawall, C. Consel, G. Muller, Towards automatic specialization of Java programs, in: Proceedings of the European Conference on Object-oriented Programming, ECOOP'99, Lisbon, Portugal, Lecture Notes in Computer Science, vol. 1628, 1999, pp. 367–390.
[71] R. Srinivasan, XDR: External data representation standard, RFC 1832, Sun Microsystem (August 1995).
[72] Sun Microsystem, RPC: Remote procedure call protocol specification, version 2, Technical Report, Sun Microsystem, 1988.
[73] S. Thibault, C. Consel, J. Lawall, R. Marlet, G. Muller, Static and dynamic program compilation by interpreter specialization, Higher-Order and Symbolic Computation 13 (3) (2000) 161–178.
[74] S. Thibault, C. Consel, G. Muller, Safe and efficient active network programming, in: 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, IN, 1998, pp. 135–143.
[75] S. Thibault, J. Marant, G. Muller, Adapting distributed applications using extensible networks, in: Proceedings of the 19th International Conference on Distributed Computing Systems, IEEE Computer Society Press, Austin, TX, 1999, pp. 234–243.
[76] E.-N. Volanschi, Une approche automatique à la spécialisation de composants système, Thèse de doctorat, Université de Rennes 1, France, 1998.